SESSION VI THE PASCAL AND C PROGRAMMING LANGUAGES: A SYMPOSIUM

GEORGE SPERLING, *New York University and
Bell Laboratories at Murray Hill, N.J., President*

SESSION VII TUTORIAL ON ON-LINE PROGRAMMING

STEVE LINK, *McMaster University, President*

Applications of multiprogramming software to real-time experiments in psychology

HOWARD L. KAPLAN

Addiction Research Foundation, Toronto, Ontario M5S 2S1, Canada

Multiprogramming operating systems are often advertised as solving the problem of competition among independent tasks operating on the same computer system. In real-time laboratories, multiprogramming systems are much more valuable for their ability to manage the relationships among asynchronous, cooperating tasks that are part of a single experiment. This cooperation allows the programming of paradigms that would otherwise require the use of faster and more expensive hardware. Examples are given from several languages and operating systems, including the small, home-built PSYCLE system and the commercially available VORTEX II system.

Three years ago, I presented a paper about the difficulties of maintaining accurate experimental timing in a multiprogramming operating system (Kaplan, 1977). Because of the existence of several levels of task priority, careless programming of one experiment's nonurgent computations can easily interfere with a second experiment's urgent input and output operations. From my arguments, Polson (1977) concluded that the difficulties of currently implemented multiprogramming systems are close to insoluble, and that experimenters would be better advised to employ single-user systems for all critical real-time applications. I want to distinguish his call for single-user single-experiment systems, which I

An earlier version of this paper was also presented at the 1979 meeting of the Canadian Psychological Association in Quebec City. I wish to thank Doug Creelman and Karen Kaplan for their suggestions, and Kathy Grishaber for assistance in preparation of the manuscript.

support, from the notion that multiprogramming has no valid uses in a real-time laboratory. What I want to present here are the arguments for another approach, the use of multiprogramming software, often designed for systems running independent experiments, to run single experiments. Such a strategy improves the ability of a computer to meet the response time demands of a single experiment, without sacrificing experimenter feedback or the complete recording of relevant data.

Many of the reasons for including a computer in a laboratory relate to its speed, but the speed of a computer system is not a single dimension. There are various components to speed—some related to computation, others related to input and output operations. One general strategy for increasing the effective speed of a computer system for some real-time tasks is multiprogramming, the division of a program into several asynchronous parts, controlled by external interrupts.

In other words, instead of specifying an exact sequence of operations that the computer must perform, the programmer specifies several related sequences, which pass messages back and forth in order to synchronize critical events within the sequences. A priority scheme, implemented in either hardware or software, then manages which of these independent programs has control of the processor at each point in time, so that the most urgent task can execute ahead of less urgent ones.

Some vendors offer both "multiprogramming," the switching of control among separately compiled programs in different host languages, and "multitasking," the switching of control among various subroutines within a single load module. Other vendors offer only one of the two options, and they are not consistent about which option they offer or the name assigned to that option. I am using the term "multiprogramming" to cover both implementations. What is essential to my arguments is not whether the components are loaded separately, but whether they can interrupt each other on an interrupt-driven, priority basis. Multiprogramming should also be distinguished from its special case, timesharing. In a timesharing system, each user is guaranteed a fair share of CPU time during each second of real-time. In the more general multiprogramming environment, one task's CPU demands may totally inhibit another task from executing.

INPUT-OUTPUT ALGORITHMS

To understand how multiprogramming software can extend a computer's real-time capability, it will be necessary to begin with a brief review of the three basic types of input-output (I/O) programming. The first, and the simplest to understand, is usually called "programmed data transfer," although the phrase "in-line transfer" is more appropriate (Figure 1). In this kind of transfer, the program code that manages the transfer is executed at those times in the program when the input or output is requested. For example, a statistical analysis program running in BASIC may need to type a number on a user's terminal. As each digit is ready to be output, the program tests the ready status of the terminal. If it is ready, the character is output and the program continues. If the terminal is busy, the program simply waits in a delay loop until the terminal is able to accept the character. No other useful computation or I/O operation can occur while this delay is elapsing. Programmed transfers may also be inefficient from the terminal's viewpoint, as the terminal can be outputting characters no faster than the program is generating them at the time. The code needed to implement this kind of output is very easy to write, and for many applications the wasted time is of no importance. However, the delays imposed by the external equipment will often make this kind of data transfer unsuitable for real-time work.

Figure 1. Flow chart for a "programmed" or "in-line" character output routine. This routine is called once for each character to be output.

The next level of complexity in I/O operations consists of interrupt-driven buffered input or output (Figure 2). A buffer is a block of memory reserved for holding data on its way in from an external device or on its way out to such a device. Under an interrupt-driven terminal output handler, the BASIC program does not always need to delay further computation if the terminal is not ready to receive the next character. Instead, the character is simply placed into a buffer, and a separate program retrieves the character from the buffer when the terminal is ready. Only when the buffer is full does the program need to wait before disposing of its current character and resuming computations. If the program is generating characters at a fixed rate, then there is no overall advantage to this method of handling output, as the program must wait for one character to be output in order to deposit each new character into the buffer. If there are continuous blocks of computation longer than the usual intercharacter times, making the output rate irregular, then there is a net throughput advantage to interrupt-driven output. The program can compute faster than the output can be printed at some times, and the output can be printing faster than it is generated at other times. The net result is that both the computer and the terminal can be kept operating at closer to their maximum achievable rates. The action of the buffer in this I/O method is analogous to that of a capacitor in filtering out voltage irregularities, or to that of a shock absorber in filtering out road irregularities. The code required to implement this kind of output handler is somewhat more difficult than the in-line code above, but the investment in programming time may provide a high payoff in operating efficiency. This is true especially if the interrupt handlers are incorporated into an operating system, so that they can be accessed from different application programs at different times.

An even more complex method of handling output is to use direct memory access (DMA) in conjunction with data buffers (Figure 3). Using this method, the fundamental unit of I/O transfer is not the individual data word, but the block, a collection of data words to be input or output as one unit. A typical block may be a line to be sent to a printer, a record to be written to magnetic tape, or 1 sec's data arriving from an analog-to-digital (A/D) converter. Each block occupies a buffer,

Figure 2. Flow chart for an interrupt-driven, non-DMA character output routine. The upper sequence is called by the application program once for each character to be output, and the lower sequence is executed once after each character's output has been completed.

and the program can be filling or emptying one buffer while the other buffer is being filled or emptied by an external device. Under ordinary interrupt-driven programming, as in the previous example, a program must be executed for the transfer of each data word. Using a DMA system, such a program needs to execute only at the beginning and the end of each block. Once data transmission has started, separate hardware takes over the job of moving each word between memory and the external device. While this does steal occasional memory cycles from the application program, this technique results in much less net overhead than does the simpler method of providing an interrupt for each data word. Because this is the most efficient data-transfer technique developed, it is the one typically used in conjunction with high-speed disks, tapes, and printers. As it requires special hardware and complex programming, it is not usually worthwhile implementing this method for low-speed devices such as Teletypes or paper tape readers. It is still not common with medium-speed devices such as CRT terminals, except in a multiprogramming environment such as a university computer center supporting many timesharing terminals.

DMA techniques are necessary because many peripheral devices require synchronous data transfers. On an asynchronous device such as a CRT, there is a minimum time allowance between characters but no maximum imposed by the device itself. On a synchronous device, such as a disk, the allowable time window for transmitting each character is very narrow, and strict timing constraints must be observed. Some inherently asynchronous devices, such as A/D converters, are useless in waveform analysis and synthesis unless controlled under strict timing constraints. It is possible for a computer to control one high-speed synchronous device at a time, but only if all other devices are left unserved during that control period. In order for the computer to do any other useful work during a data transfer with

a synchronous peripheral, auxiliary hardware must maintain the device's timing and access memory directly, without CPU intervention between successive timed data points. With such hardware installed, the CPU's remaining interactions with the peripheral may be handled at any time after the device is ready, that is, asynchronously.

Block data transfers may improve the efficiency of I/O, even when they are not initiated on an interrupt-driven or DMA basis. In the Commodore PET, for example, cassette data are transferred in blocks of 192 characters in order to reduce the overhead of starting and stopping the tape motors. Even though most characters can be written to the tape buffer with no I/O wait, the total I/O time remains proportional to the number of characters transferred, regardless of the irregularity of computational and I/O demands, because no computations can occur while the tape transfer is happening. While there are interrupt-driven facilities within the PET, the standard software provides no mechanism for their use in smoothing I/O operations to increase throughput.

The important differences among the three I/O methods can be summarized as follows: In programmed (or in-line) data transfers, waiting for external devices is an activity that alternates with the computational program and with waiting for other devices. In ordinary interrupt-driven transfers, I/O activity with very low CPU demand can essentially run in parallel with CPU activity or with other I/O activity. However, as each data transfer requires the suspension of one ongoing program to invoke another, the context-switching overhead of handling many devices or fast devices can become significant. To reduce this overhead, DMA transfers can reduce the number of interrupts that need to be handled by letting hardware replace software for the transfer of individual data words, and they can maintain synchronous transmission on several peripherals simultaneously.

Figure 3. Flow chart for an interrupt-driven, DMA block data output routine. The upper sequence is called by the application program once for each data item to be output, and the lower sequence is executed once after each buffer's output has been completed. In general, the lower routine executes much less frequently than does the upper one.

THREE WAYS FOR A CPU TO BE TOO SLOW

In order to understand the circumstances in which multiprogramming can and cannot increase a computer system's capacity, we should begin by considering a very straightforward biofeedback experiment (Figure 4). The only variable stimulus is a light, and the only response being collected is an EEG channel. The EEG is recorded in 1-sec segments, and the computer is required to perform a statistical manipulation (such as a Fast Fourier transform) on each such 1-sec segment. At the beginning of each segment, the computer must briefly send a signal to initiate collection of that segment's data and to update the stimulus light based on recent EEG analysis. Once the input begins, DMA hardware brings in the data segment with little impact on whatever other computations are occurring during that second. The process of starting each segment consumes only 1 msec, so that 99.9% of the processing capacity is available for the EEG analysis itself. We will assume, however, that the computations required to process each segment of data take 1.5 sec. In other words, the data cannot be processed as fast as they arrive. If we need to process all of the data in real-time as it is collected, we cannot do it. What is lacking is overall throughput capacity, or the ability to perform a sufficient number of computations/sec. What is needed is a faster computer or a simpler computational algorithm—multiprogramming cannot help.

Let us relax the requirements (Figure 5). We will still record all of the incoming data, but we will analyze only alternate segments' data and update the stimulus light 1 sec after each analyzed block has been completely input. Here the problem is no longer throughput capacity, as we need only 45 sec of computation for each minute of data collection. Instead, the problem is peak

Figure 4. First biofeedback example. Each asterisk indicates a time when the feedback light must be updated. Because each data block requires more time to process than to input, the computer falls increasingly behind. This is a total throughput capacity problem that multiprogramming cannot aid.

Figure 5. Second biofeedback example. Each analyzed data block's results are needed before they can be completed, because the computer's total throughput capacity cannot be converted to peak response capacity.

response capacity, for the required computations cannot be performed fast enough after each triggering event, the completion of one segment's input data. Again, this is not a case in which multiprogramming techniques can help. However, if we could somehow arrange to begin computations on each block of input data before that block has been completely collected, then it might be possible to implement this paradigm.

If we relax the requirements even more, still analyze alternate blocks, but update the light 2 sec after each analyzed block has been completely input, the situation is different (Figure 6). Here, both the throughput capacity and the peak capacity are adequate, as we need only 1.5 sec to generate a result that is needed after 2 sec. The only problem is that two-thirds of the way through the computation, we must temporarily suspend computation in order to send the urgent signal that initiates input of another segment's data. In other words, the problem is not in finding enough computation time, it is in finding a block of computation time that will not interfere with I/O activity. If we can arrange to briefly suspend or interrupt the computations, we can ensure the recording of all of the data. We connect our 1-sec timer to the computer's interrupt to divide this task into two parts, foreground and background. The foreground task occurs once every second, consumes only 1 msec, and updates the stimulus light and the response recorder. The background task is no less important, only less urgent. It can safely be interrupted by the foreground task and still complete in time for the start of each 3-sec block of recording and analysis.

This is one of the simplest of multiprogramming situations. There are only two tasks, and there is very little timing information to be communicated from one task to the other. In this task, we could almost dispense with the interrupt structure. If we could determine while doing the computations when exactly 1 sec had

Figure 6. Third biofeedback example. Although the time needed to process each block is not available in one uninterrupted interval, the operating system makes parts of two intervals available and the computer is then fast enough to perform the task.

passed, based on the number of iterations performed in the analysis algorithm, we could use that as the signal to break for 1 msec and update the input recorder. However, in just about all practical situations, the computational tasks and many of the I/O tasks take variable amounts of time to complete, and the external time cannot be predicted from the number of internal operations completed. Instead, we must rely on external hardware, such as counters, timers, and the ready status indicators of tape and disk systems to indicate when to switch from one task to another. In general, it is more efficient to let the most important of these external signals interrupt the computer, rather than having the computer test their status as part of ongoing operations. In other words, the continual monitoring of these statuses should be left to hardware, the interrupt system, rather than included in the software of the application programs.

FOREGROUND-BACKGROUND MULTIPROGRAMMING SYSTEMS

Although multiprogramming systems may involve many priority levels and task streams, it will be useful for the moment to consider just two levels, foreground and background. The foreground level includes all events whose timing must be controlled exactly (within the grain of the real-time clock) in order to satisfy the experimental protocol. This includes the presentation of stimuli and the collection of responses. The other necessary activity, including nonurgent decisions about future stimuli, monitoring for exceptional conditions, passing data to disks or tapes, and communicating with the experimenter, is considered the background task. This difference is not identical to the difference between I/O activity and CPU activity. The foreground must always involve I/O, because it is concerned directly with control of real-time real-world events. Those computations most directly and urgently concerned with the

foreground I/O may also be considered part of the foreground. The background may consist entirely of CPU activity, for example, when the background task simply computes the next stimulus level for the foreground to implement. The background may also include substantial I/O, such as transferring completed data blocks to or from disk because not all of the data can fit into memory at one time. This I/O is classified as background activity because the timing constraints are ranges of time, rather than particular moments in time. Each data block must be written to disk before the memory it occupies is overwritten by future data, allowing a substantial margin for exactly when those data are written. The collection of that data block, however, must have been initiated at a fixed time relative to its eliciting stimulus situation; therefore, the data collection is considered part of the foreground activity.

When is a paradigm suitable for a multiprogramming approach? In general, multiprogramming will be useful when the amount of activity scheduled between significant events of a single task stream is too variable to allow that stream to maintain timing accuracy. If there is not enough time for essential activity between external events, then there is a peak response problem, as in the second biofeedback example, and multiprogramming cannot help. But if we have many foreground events whose completion does not depend on the immediate state of the background activity and only occasional foreground events dependent upon the completion of particular background events, then we have the prototypical multiprogramming situation.

The essence of real-time multiprogramming systems is the sending of messages among tasks operating at different priority levels, foreground and background. For example, if a foreground program under severe real-time constraints needs to send a printed message to an experimenter, it does not execute a routine in which each character is sent only when the terminal is ready to receive it. Instead, it passes the characters, either one at a time or as a completed buffer, to a background program that can safely wait during the intervals when the terminal is busy. In the general situation, the various subtasks of a multiprogramming system pass messages to each other through shared memory locations. Such a shared location may contain the serial number of a message to be printed, may be a pointer to a text buffer to be printed, may contain the serial number of the last block of data collected or processed, or may contain any other information needed by one subtask to synchronize its activities with another subtask. Dijkstra (1968) introduced the term "semaphore" for such a shared memory location. Although he proposed a set of only two primary synchronizing operations by which competing tasks could inspect and update these shared locations, the term is often used more generally for any set of rules for inspecting and resetting a shared location to synchronize processes (Lee, 1976).

CIRCULAR QUEUES

The circular queue is a data structure of fundamental importance in multiprogramming systems (Figure 7). A circular queue, or circular buffer, is also known as a first-in, first-out list. One program installs data in this list, and a second program removes data from the list, in the same order in which it was installed. At least one of the two programs is interrupt driven. The queue is called "circular" because the same memory locations are reused to hold different data items as the programs progress. A pointer is used to fill the queue, and a separate one to empty it. When the pointer passes the last item, it is reset to access the first item again. It is possible for the data-removing process to fall as far behind the data-installing one as the length of the queue. We will see some of the possible consequences of this event, when the queue is full and can hold no more data, as we look at some representative circular queues in multiprogramming situations.

The simplest circular queue occurs in systems such as FOCAL or BASIC, as implemented on the PDP-8 (Figure 8). This queue manages the sending of characters from the computational program to the terminal and is an elementary form of output spooling from a background program. If we consider the ultimate goal of running such a program to be sending output to the terminal as fast as possible, then we can consider the terminal-handling software to be the foreground program and the computational part that feeds it to be the background. Whenever the background program needs to print a character, it sees whether the terminal is busy or idle. If the terminal is idle, it sends one character and sets the status to busy. If the terminal is busy, it installs the character at the next free location in a circular buffer. As the terminal becomes idle again after printing each character, the foreground program removes characters from the buffer and sends them to the terminal until the buffer is exhausted. If the buffer is full when the background has another character ready to install, the background simply loops, waiting for an interrupt to occur and the foreground to remove one character from

Figure 7. A circular queue. Separate pointers are used to store and to retrieve data, under the asynchronous control of two tasks.

Figure 8. A circular queue used as a terminal output buffer. Such queues are used, for example, in PDP-8 FOCAL and BASIC.

the queue. Since the background exists only to feed characters to the foreground, this looping delays the CPU but is otherwise harmless to the ultimate goal of the program.

If the background program also requires input from a terminal or a discrete-character input medium such as paper tape, then a similar queue can be used to buffer input characters. Such a queue is used in the FRIVLOS operating system on the PDP-12 (Figure 9). As each character is input from the external reader, it enters the queue, and as the background program needs each character, it is removed from the queue. In this case, the consequences of potential buffer overflow are more serious. If the input is being typed at a keyboard, the possibilities include ignoring all characters typed after the buffer is full, issuing a fatal error message and terminating the background program, or issuing a warning signal with bell codes or other nonprinting characters. If the input is arriving from paper tape, the program can send a signal to disable the reader until the buffer again has space in it. In practice, the resumption of paper tape input might not occur until the buffer is at least half empty, to avoid a large number of reader-start and reader-stop operations.

For both input and output operations, the justification for the somewhat complex interrupt-handling structure is the irregular rate at which the background generates and consumes I/O characters. If a fixed amount of computation occurred between characters, then there would be no throughput advantage to the interrupt-driven output system, as the output rate would be limited by the terminal or the program, whichever is slower. With the interrupt-driven output handler, bursts of characters can be generated at one time, such as in the conversion of a result from internal binary to external ASCII, and the time during which they are actually being printed can be utilized for a burst of CPU

Figure 9. A circular queue used as a terminal input buffer in PDP-12 FRIVLOS. The user can type ahead of the explicitly requested input, up to the capacity of the input buffer.

activity, such as calculating a complex formula. The larger the circular buffer included in the system, the longer the range over which CPU and I/O activity can be averaged to provide smooth, efficient throughput.

Either the contents of the queue positions or the values of the pointers can be used to indicate which positions are occupied. As an example of the first method, the program that removes characters from the buffer can set each just-vacated location to 0, indicating that it is empty, providing that 0 is not the binary value of any valid character code that can fill the location. The other method requires comparison of pointers. If the filling pointer catches the emptying one, the queue is full; if the emptying pointer catches the filling one, the queue is empty. All other circumstances mean that the queue is partially full. In a variant of this method, both the filling and the emptying processes increment counters of queue elements stored or retrieved, and the counters are not reset when the end of the list of positions is reached. Instead, modular arithmetic is used to convert the counters to pointers into the queue. For example, with a 10-item queue, the last decimal digit of the serial number of each item stored or retrieved indicates which queue element contains the item. If an emptying process's counter fails behind the filling counter by more than the queue length, then overflow has occurred, and the emptying process can no longer retrieve valid data from the queue. However, if a faster emptying process has been copying the data to auxiliary storage, the slower process can use that back-up store to obtain the necessary data.

An extreme variant of the circular queue is the double-buffered I/O system found on many computers (Figure 10). This buffering is standard for programs running on IBM 370-series hardware under the familiar MVT or MVS operating systems. In such a circular

queue, the items being stored and retrieved, at least from the standpoint of the interrupt-handling I/O drivers, are not single data words, but instead, they are entire blocks or buffers of data to be transferred between the computer and tape or disk storage, using a DMA transfer. The "circular" property here becomes a simple alternation between buffers, and each buffer is either ready or not ready for use at any time.

Such alternation between buffers is a critical component of many real-time data-logging routines. In order to achieve continuous input of large amounts of data, such data must be written to back-up storage while subsequent data are being collected. For example, in digitizing whole sentences at 20,000 samples/sec, 10 sec worth of speech would occupy 200K worth of real memory, well beyond the capacity of most laboratory computers. However, it is quite practical to read alternate 200-msec data records into each of two 4K buffers and to dump each buffer to disk or tape in the 200 msec while the alternate buffer is being filled. Such a procedure requires that both the input and output devices work on a DMA block-transfer basis, and that the worst-case disk or tape access time be less than the block duration. If the average output time, but not the worst-case time, is less than the input duration, then it is still possible to spool the data to disk, using more than two memory buffers to hold the additional backlog. Disk buffers can also hold a backlog for even slower or more irregular tape backlogs, leading to a very general spooling system (Kaplan, 1978).

A circular queue might also be part of a system to generate random music, where the peripheral devices include a function generator and a clock (Figure 11). The background task installs three words at a time into the queue, representing the duration, frequency, and amplitude of the next note to be played. Rather than interrupting the computer at a fixed rate, the

Figure 10. Classical double buffering, a limiting case of the circular queue.

Figure 11. A circular queue used in a random music generation system. Because the clock limits the rate at which the new note parameters are needed, the system is not so much I/O-bound as I/O-relieved.

clock can be reset for each note played, interrupting only at the end of each note or internote gap. Upon each interrupt, the foreground task resets the note-duration clock, terminates the current note, sets up the frequency and amplitude of the function generator, lets the internote interval complete, and initiates the next note. At that point, it returns control to the background program, which resumes filling the queue as fast as spaces in it become empty. What makes this program interesting is the possibility of highly variable note durations. If all notes were of equal duration, and if the process of selecting each note consumed a roughly constant amount of CPU time, then there would be no need for the queue. However, with the queue, only the average rate of selecting notes need match the average rate of playing them. During the playing of fast passages, the selection process may generate fewer notes per second than are output. The circular queue, which contains a backlog of notes already selected but not yet played, will keep the function generator supplied with information until the tempo declines again and the background program can catch up with the foreground rate.

ASSUMPTIONS MADE IN DESIGNING OPERATING SYSTEMS

The music-playing example demonstrates the need to rethink some of our usual vocabulary in discussing computer systems. We often talk of systems being "CPU-bound," meaning that computations cannot keep up with the I/O, or "I/O-bound," meaning that CPU activity must cease while I/O activity completes. In the music system, it would be more appropriate to describe the background task as "I/O-relieved" than as I/O-bound. If we think in large-system terms, where we want

to maximize the number of calculations per second, then the system is I/O-bound. But if we think in terms of servicing the foreground, then it is I/O-relieved, in the sense that the background is not required to produce more calculations per second than the foreground can use. We are only wasting CPU cycles, a situation that many commercial operating systems are designed to avoid.

This rethinking of the role of the CPU and the I/O systems is important, because many commercial multiprogramming systems stem from assumptions unlike those of laboratory multiprogramming systems. Multiprogramming developed in the days of very expensive CPUs that, while slow by today's standards, were still fast compared to the I/O devices attached to them. In order to make use of the CPU time left over while waiting for one task's I/O to complete, the CPU activity associated with other tasks could be executed. Operating systems such as HASP and JES2 on IBM equipment, RSTS on DEC PDP-11s, and VORTEX II on Sperry-Univac minicomputers are of this type. The stated purpose of these operating systems is to protect the independent tasks from each other, giving each task the illusion of a complete computer to itself, while sharing the limited resources of an expensive CPU and its peripherals. In the laboratory a multiprogramming operating system is, instead, often needed to manage the overlapping and related activities of several highly dependent tasks.

While many laboratory operating systems are suitable for the kind of shared-process foreground-background multiprogramming described here, the literature supplied by the vendors obscures that fact. It is typical for the priority structure to be described in terms of important and less important control tasks with different priorities. For example, in a physiological laboratory, monitoring EKG signals for signs of shock or other physical danger has a higher priority than monitoring eye blinks to detect successful classical conditioning, but the two monitoring tasks are relatively independent. If the facilities for sharing information among separate tasks are difficult to use, time-consuming, and limited to small amounts of information, then sharing processing between foreground and background components of a single experiment will be difficult. However, if different tasks can simply share large blocks of memory suitable for both one-word semaphores and extensive data buffers, then this powerful style of programming may be easily implemented. We can now take a detailed look at two separate complete experiments implemented on multiprogramming software to see how circular buffers and priority structures are used to manage the flow of information. The first system is an auditory threshold experiment run under the home-built PSYCLE system on a small PDP-8/S, and the second is a visual pursuit experiment run under the commercially available VORTEX II system on a V76.

MULTIPROGRAMMING UNDER PSYCLE

The original PSYCLE system was developed by C. Douglas Creelman and some of his students at the University of Toronto in the late 1960s. The system runs on a PDP-8/S computer with 4K by 12 bits of core memory, a Teletype, an audio recorder for program and data storage (similar in concept to the cassette recorders used on today's low-cost microcomputer systems), and various equipment for controlling auditory stimuli (function generators, filters, audio switches, etc.). I implemented the current, multiprogramming version in 1972. Although a rudimentary compiler is available, most applications are coded in assembler, as was PSYCLE itself. In the experiments discussed here, the operating system was used to run a procedure known as Multiple PEST. PEST stands for "parameter estimation by sequential testing" and refers to an algorithm for finding auditory thresholds in two-alternative forced-choice experiments, where threshold means whatever stimulus level is necessary for 80% correct performance. The PEST routine adjusts the current testing level for each stimulus on the basis of past performance at various testing levels. The term "multiple" means that this adjustment is performed separately for each of three subjects and for each of a set of alternative stimuli differing in a characteristic such as frequency or melody.

A typical experimental session is divided into 10 runs, each consisting of 100-150 trials in which three subjects are tested in overlapping alternation. That is, on each trial, all three subjects receive separate first-stimulus presentations, then all three receive separate second presentations, and then all three signal their responses. The timing of the experiment thus requires a mixture of clock intervals ranging from a few milliseconds to about 2 sec. During the course of the experiment, it is very useful (although not absolutely essential) for the experimenter to know the current testing level on each of the possible stimuli that might be presented. This information can be obtained in two forms, either a message every time a testing level changes or a complete matrix of testing level by stimulus type by subject, produced on demand. The first type of list is most useful if changes are printed as they occur. While the complete matrix can be produced at the end of each run, the relatively slow speed of the Teletype makes it preferable that the printing of the matrix begin somewhat before the run ends, even though this means that the information is not totally current. As this printing is primarily for feedback to the experimenter, and not a substitute for saving the data in a more careful format, such compromises are considered quite acceptable.

The problem with printing the data during the experiment is that a typical message can occupy at least 20 characters, including the trial number, subject number, stimulus number, and new testing level. As this amount

of printing can occupy 2 sec on a Teletype, and as such a message could theoretically be required for all three subjects, there is not enough time to print this message during any of the timed intervals that are part of the experiment. In addition, the time required to convert the information from its internal, highly condensed format to decimal numbers suitable for printing might consume significant CPU time on this rather slow computer. Therefore, this experiment was conducted under a three-level priority system, PSYCLE (Figure 12).

At the highest priority level, or foreground, is the logic required to conduct the trials, including control of stimuli, collection of responses, and decisions about which stimulus level to test next. Whenever the testing level changes, the foreground program quickly writes a few words into a circular buffer, indicating the trial, the subject, the stimulus condition, and the new testing level. The inherent logic of the PEST procedure makes it quite improbable, but not impossible, that this buffer will be completely filled. During the execution of these foreground tasks, the interrupt system is disabled. As soon as the urgent tasks are complete, the interrupt is reenabled and control is passed to the background loop. This background loop continually scans the changes-of-level buffer filled by the foreground, and it also scans a location containing the last Teletype key pressed during the experiment. If the changes-of-level buffer has any information in it, that information is translated to printing format and sent to the Teletype via a second circular buffer, similar to the one used in FOCAL or BASIC. If that buffer becomes full, the background can safely wait for space to appear in it, without interfering with the foreground program. The changes-of-level information is printed on the left side of the page. Meanwhile, if the operator presses a specified key, or if the foreground program emulates the pressing of that key by putting its ASCII code into the last-key-pressed location (to start printing an additional matrix halfway through a run of trials), the background program begins printing the complete matrix of current testing levels on the right-hand side of the page. The background loop

Figure 12. Information flow in PSYCLE. Based on interrupts from the clock and Teletype, the dispatcher controls which of three processing levels will execute at any time. In the lower left corner can be seen the subjects' contribution to this process.

can be interrupted by both the foreground program, to conduct the next part of a trial, and the midground Teletype handler, to fetch the next character from the second circular buffer and send it to the printer. The Teletype handler itself can be interrupted by the foreground clock, because it requires a nontrivial amount of time to fetch a character and update the buffer pointer.

MULTIPROGRAMMING UNDER VORTEX II

The other example of a multiprogrammed experiment is rather more complex, runs on much higher speed equipment, and executes under a commercially available operating system. The hardware configuration includes a Sperry-Univac (formerly Varian) V76 computer with 128K by 16 bits of semiconductor memory, a hardware floating-point processor, two disk drives, nine-track magnetic tape, and A/D and digital-to-analog (D/A) converters. The disk drives, the magnetic tape, and the A/D converter can independently and simultaneously operate under DMA control. The system is part of the Human Responses Laboratory at the Addiction Research Foundation of Ontario. The operating system, VORTEX II, allows programs to operate in independent memory and also to access a shared memory area known as foreground blank COMMON. In typical real-time experiments, COMMON is used to hold both semaphores, one-word locations used by different tasks to advise other tasks of their progress, and data buffers that are filled and emptied by the various foreground and background tasks. While some system functions need to be coded in assembler, the majority of applica-

tions programs are written in FORTRAN IV and processed by an optimizing compiler.

One experiment involves visual pursuit, with a subject trying to track a point cycling horizontally across the face of a CRT under computer control. Each position of the point is generated by a D/A converter, which sends a new voltage (under program, not DMA, control) every 10 msec, or 100 times/sec. Eye position is transduced by infrared sensors, converted to a voltage, and returned through the A/D converter. Although the input voltage is reasonably linear as a function of eye position, we do not know the calibration function. To start the trial, the point is making one sinusoidal circuit every 4 sec (Figure 13). Once the computer determines that the subject is tracking properly, the point begins to accelerate, so that 21 sec later it is making one cycle every .5 sec. During both the constant and accelerating portions of the trial, data blocks consisting of each second's data are read into a pair of buffers in foreground blank COMMON, using the standard double-buffering algorithm. To conduct this experiment, the computer's priorities are to capture all of the input data, to transfer all of these data to disk and to tape for further analysis, and to use the 4-sec cycles to calibrate the input sensors and decide when the stimulus is being tracked properly, given that tentative calibration, so that the acceleration of the stimulus may begin.

The highest priority task is the one that services the A/D converter, operating under DMA control and providing eye position data (Figure 14). As the system clock indicates that each block of input data is about to complete, this program takes control of the computer, waits for the block to end, immediately outputs the

Figure 13. Stimulus position during the calibration phase of the visual pursuit experiment. The 4-sec sine wave is repeated until the I/O voltage correlation is sufficiently high, and then the stimulus is allowed to accelerate. The first 430 msec of the acceleration portion are identical to the first 430 sec of the sine wave, allowing time to make the calibration decision in a background task.

Figure 14. Information flow in VORTEX II during the visual pursuit experiment. Note that there are three separate tasks retrieving data from the same circular buffer. In addition to the tasks shown, the dispatcher may activate editors or compilers during the unused portions of each 10-msec system clock period.

address of the next buffer into which data are to be received, and restarts the converter. This restart operation takes less than 10 msec, so that in every 10-msec interval, including the one that terminates a data block, the next highest priority program can send the appropriate stimulus voltage via the D/A converter. In addition, even though the current block of input data may not be completely collected, those input voltages corresponding to the last voltage sent are already available in memory, and so a counter of data points received is also kept in COMMON, along with the data buffers themselves. When the calibration is acceptable, the changeover from constant to accelerating sinusoidal motion is made 430 msec into the current cycle, when the stimulus is at its rightmost peak, its velocity is lowest, and the change can be made most smoothly.

A CPU-bound, lower priority task, which we can consider to be a background task, inspects the eye position data as they arrive in COMMON. Monitoring the count of data points received, the task waits for each consecutive point to appear in shared memory and then incorporates it into sums and sums of squares for computing the correlation between the output voltage sent and the input voltage received. After 400 points have been included in these sums, the background program calculates for each eye the correlation coefficient and an equation relating input voltage to presumed eye position. If both correlations are at least .95, then this background program sets an indicator in COMMON so that the foreground program can begin to increase the frequency of the driving sinusoid. This signal is not needed until 430 msec after the end of the 4-sec cycle, but during those 430 msec the beginning of the next cycle is being output, in case it is needed for calibration.

If the calibration attempt must be repeated, the background program resets its sums and begins to calculate the correlation for the next 4-sec cycle, until calibration is acceptable. The working assumption here is that the correlation cannot become high unless the subject is following the stimulus fairly accurately, and that any consistent angular bias will be maintained over the accelerating portion of the trial.

Once the stimulus begins to accelerate, the background task switches its function to evaluating the quality of the subsequent tracking as a function of driving frequency. The details of these calculations are not important here, but again, data are retrieved from the circular buffers in shared memory as they become available, so that these summary statistics are almost completely calculated when the 21-sec trial ends. While these interesting but not urgent computations are progressing, higher priority tasks (but not as high as the task that generates the stimulus) are copying the data from the shared buffers to disk and to tape. In this case, we have one program filling a circular buffer and three separate programs retrieving data from the buffer, effectively at the same time. The slower processes, the ones writing to tape and calculating statistics, can obtain data from disk if the copy in COMMON has been overwritten with newer data.

If we had enough memory to store all of the input data until the 21-sec response had been collected, then we could avoid the multiprogramming aspects of data storage and simply write the data to tape and disk at the end of the trial. Furthermore, the calculation of statistics about the goodness of the accelerating tracking is only interesting, not essential, in real-time. Where the multiprogramming aspects of VORTEX II become absolutely essential is in the calibration stage of the trial, when sums and sums of squares are being built as each data point arrives in memory. After almost every data point, the first 399 of each block of 400, the necessary calculations take a reasonably fixed amount of time, updating several sums and sums of squares. After the 400th point, however, additional calculations must be performed, involving several more multiplications, divisions, and perhaps square roots. If the correlations are not acceptable, then more time must be spent in reinitializing the component sums and sums of squares. Without multiprogramming, the processor would have to be fast enough to allow the regular updating of sums, the testing of the final correlation, and the resetting of the sums in the time between any two stimulus points being sent. With the multiprogramming software available, the process of rejecting one cycle and preparing for the next cycle can be allowed to overlap the collection of several input voltages, providing only that the background program can make up the backlog later in the cycle. In other words, the multiprogramming software is increasing the effective peak response capacity of the

computer, allowing calculations that would otherwise need to be completed in a very short time frame to be spanned over several time frames instead.

MULTIPROGRAMMING IN BASIC

With the current proliferation of small, inexpensive computers offering BASIC, it becomes important to consider how the principles of multiprogramming as expressed here can be implemented in such a system. The need for such an implementation is clear, as the overhead imposed by operating in BASIC can convert a relatively fast hardware configuration into a relatively slow, though powerful and convenient, software configuration. Algorithms that can easily run in the foreground of an assembler-based or compiled system may need to be deferred to the background of a BASIC-based system, because their execution time interferes with critical real-time tasks. Unfortunately, many BASIC configurations, particularly those based on read-only memory, do not allow the interrupt-handling software to be modified conveniently, as would be necessary to implement multiprogramming if not already present.

In discussing the possibilities of an interrupt-driven BASIC system, we must distinguish between two kinds of interrupt handling. One kind interrupts BASIC to execute non-BASIC code. An example of this occurs on the Commodore PET, where the hardware clock interrupts the processor 60 times/sec and the software clock is updated on each interrupt. At the time of the interrupt, a few registers' values need to be saved and restored, but none of the memory used by the BASIC interpreter itself need be disturbed. Similarly, the keyboard is scanned on interrupt to store data into an input buffer, from which characters are retrieved as needed. Although it is not part of the standard software, a similar routine could be written to serve a printing terminal with characters stored in a memory buffer. For many types of experiments, this kind of interrupt servicing is all that is required. If each interrupt requires some fairly simple and rarely changed sequence of instructions to be executed, then even hand assembly of the instruction code and its installation with POKE commands are feasible.

As an example of such a mixture of BASIC and assembler, consider the problem of monitoring a rat on a fairly rapid fixed-ratio schedule and of using the CRT screen to emulate a cumulative recorder at the same time. Without the emulation, it is probably within the capacity of most versions of BASIC to detect each barpress as emitted, to count the presses, and to initiate the reinforcement. It is also within the capacity of about 30-40 machine language instructions to implement the same paradigm, providing that all of the overhead, such as asking the experimenter what ratio to use that day, is taken care of in BASIC. Therefore, one rational strategy is to write the time-critical code in assembler,

letting it operate on an interrupt-initiated demand basis and letting it pass messages to and from BASIC in a few shared memory locations. These locations might be configured as a 2 by N matrix with each pair of locations containing one for clock time at barpress and one for whether or not the response was reinforced. The BASIC program could then monitor this array, watching for previously unrecorded responses and taking whatever time is necessary to calculate the screen coordinates and to produce the graph. By thus dividing the complex time-consuming parts of the paradigm from the simple, rapid, easily programmed ones, the latter being programmed in assembler, the computer is allowed to provide feedback that would normally be beyond the capacity of a BASIC-implemented real-time system.

The other kind of interrupt handling in BASIC involves the writing of the interrupt handlers themselves in BASIC. While this allows for the implementation of fairly elegant and complex interrupt-processing code, it suffers from very severe timing constraints. Only fairly fast processors or fairly slow paradigms can make use of this kind of interrupt handling. The problem arises from two sources. One source is that the interrupt-handling code itself, being written in BASIC, cannot execute very fast, and the elapsed time between the initiation and the completion of interrupt service may be too great. This is true particularly when the interrupt is used to initiate a complex decision and the computer's response cannot be made until that decision is completed. A more fundamental problem arises in the time required to initiate interrupt handling. When the routine to be initiated on the interrupt is not itself coded in BASIC, only a few registers need to be saved and restored when allowing the service routine to execute. However, if the routine is executing in BASIC, then much more volatile information (line pointers, floating-point accumulators, return addresses, etc.) needs to be saved. In order to avoid that problem, many versions of real-time BASIC only record, but do not process, interrupts as they occur. The actual processing is deferred until the completion of the current line of BASIC code, so that none of these auxiliary storage locations contains any unique information, that information having been transferred to program control and data variables.

"End-of-line" interrupt handling severely limits the response time of the system. In particular, a response is not guaranteed before the execution time of the slowest line of the interruptable program. If that line involves a complex matrix or graphics operation, then interrupt processing may easily be delayed for up to 1 sec or so, even on a relatively fast computer. In order to achieve an acceptable response time, it is necessary to divide the interruptable program into very short statements, even when longer statements would process the data more efficiently, in order to limit the maximum time before

interrupts are handled. Because of the difficulties of guaranteeing that a supposedly low-priority background routine will not interfere with responding to a high-priority interrupt, BASIC systems featuring end-of-line interrupt handling cannot be generally recommended for high-speed real-time experimentation. Where such systems are purchased for other advantages, the interrupt handling must be carefully planned and tested in each experiment, in order to verify that response time constraints are met.

Given the obvious disadvantages of end-of-line interrupt handling, we might wonder why that is the only kind allowed on many systems. The reasons are both technical and economic. In order to let one line of a BASIC program interrupt another line in progress, it is necessary to have multiple sets of critical memory locations: floating-point accumulators, character pointers, text buffers, and so on. One way to implement these is with memory mapping hardware, but that involves a cost and complexity beyond most low-cost systems. Another way is to access these critical memory locations through index registers, rather than directly. While such an approach is quite practical, it would require more systems programming effort and cost on the vendor's part than many of them are willing to undertake, and it might also make single-task benchmark programs run slightly slower, possibly scaring off some unsophisticated users interested in a "fast" computer system. Even were such an approach adopted, it would still not be possible to interrupt the program after any one machine instruction, because operations such as updating a multiple-byte number would have to be completed as a unit. However, taking such an approach would reduce the maximum time before processing an interrupt from the order of many milliseconds to a few microseconds and allow the programming of very precise experiments in BASIC. Without such a feature, the microsecond-accuracy clocks being included in some low-cost systems are next to useless.

Beyond the problem of shared critical memory is the problem of shared I/O resources. It is clearly unacceptable for one high-priority program segment to begin writing to the CRT while another segment has an output line in progress. Instead, all program segments should handle shared I/O devices by passing messages to and from a monitor routine. In this way, all CRT, keyboard, and disk transactions would be complete lines or blocks of data. These problems of I/O contention are by no means peculiar to BASIC; their effective resolution is one of the main functions of software such as VORTEX II and RSX-11D. They are mentioned here only to emphasize that implementation of a complete multiprogramming system within BASIC is a formidable task, one which far exceeds the usual demands of what began as a hobby market. However, there may well be a market for a compromise multiprogramming BASIC system, in which interrupt-initiated subroutines may

execute I/O only on nonshared devices, all shared-device I/O is handled by the background program, and interrupt handling is not deferred to the end of the current line.

EVALUATING MULTIPROGRAMMING SYSTEMS

Many of the considerations for selecting among competing multiprogramming systems are the same as in any computer-selection process: adequacy of documentation, reliability of local repair service, quality of code produced by compilers, ease of use of editors and utility programs, and so on. In addition, some other considerations are peculiar to the multiprogramming approach to laboratory control. These considerations include questions of timing accuracy, interrupt structures, memory management, and related operating system features.

(1) What is the maximum time interval for which the operating system can delay processing of an urgent interrupt? During the process of switching between tasks, or of executing user task requests for operating system intervention, the interrupt system may be disabled for periods ranging from a few microseconds to several milliseconds. This disabling allows shared but nonreentrant code to execute, without critical return addresses or memory locations being altered by further interrupts. In general, this is a rational way to write an operating system, but it may not allow a sufficiently small interrupt response time. It is especially deceptive for manufacturers to advertise the possibility of "directly connected" interrupts, ones that activate code while bypassing the dispatcher overhead, on systems that require the interrupts to be periodically disabled for nontrivial durations.

(2) Can memory be shared conveniently only among tasks compiled separately, among tasks compiled together, or both? In a language such as PL/I or BASIC, it is easy for subprograms to reference the same memory locations using the same names. This makes it quite convenient for those subprograms to use such named locations as semaphores and to use named arrays as shared memory buffers. In FORTRAN, shared locations must be declared in COMMON or passed to subprograms as explicit parameters. If the routines are not processed by one invocation of the linking loader, then these COMMON blocks must reference a single fixed-address memory buffer in order to be truly held in common among separate tasks.

Operating systems differ in whether the several asynchronous tasks being managed must be, may be, or cannot be part of a single memory partition created by one link loading. Allowing them to be subtasks of one partition may ease memory sharing problems, but it may also require the recompilation of large amounts of code when only one module is changed. The alternate approach, allowing only one task per load module (or

per memory partition), allows much more independent development of the modules but may make the sharing of information among them more difficult.

Some systems attempt to ease the memory sharing problem by establishing a specific protocol for sending messages between tasks in separate memory partitions. In general, though, the time taken to execute such protocols is an order of magnitude above the time to simply dump data into truly shared memory, and this may be too slow for high-speed experiments. Among solutions truly sharing memory, neither the separate nor the combined compilation approach is inherently better, but one or the other may be considerably more consonant with a particular user's style.

(3) Can programs that are not subtasks of a single main program ever execute concurrently? On one level, this can be seen as asking whether an editor or compiler can ever run in the background during a real-time task. On another level, it is also a question of how much advance planning needs to be done in deciding which tasks will execute concurrently. If, at any one time, only two of five tasks will be active, there is a great difference between requiring memory for only two at once and requiring memory for all five to be simultaneously loaded, with any pair executing concurrently.

There is a similar question about whether tasks in two different host languages can run simultaneously. In theory, a fast FORTRAN program to run an experiment, leaving data to be inspected by a slower but easily developed BASIC program for experimenter feedback and statistical analysis, is a rational way to program some experiments. However, if the computer system requires the same memory for the FORTRAN and BASIC run-time routines, then the combination is not practical.

(4) Can dormant tasks prevent themselves from being swapped out to disk when inactive? In a time-sharing environment, it is reasonable to move a dormant user's memory image to disk when no character has been transmitted recently. In a real-time environment, it is inappropriate to remove programs having infrequent but urgent demands from memory. Multiprogramming systems that emphasize saving memory by swapping out inactive tasks are particularly poor candidates for laboratory systems. Although provisions may exist for locking dormant tasks into memory, the operating system may be optimized under the assumption that such locks are not occurring, and the resulting memory shortage for other tasks may seriously compromise their response time.

(5) Are device drivers tasks with user-settable priorities, or do all drivers run ahead of all user tasks? In a commercially oriented timesharing system, where total throughput is a goal, it may be reasonable to do every I/O operation as soon as the device and memory buffer are ready. In a real-time laboratory, I/O for the background part of a multiprogrammed task or for a low-

priority editor should not be allowed to inhibit the CPU demands of a critical real-time task. With a high-speed CRT generating 960 interrupts/sec, the CPU time to recognize and service those interrupts may be quite significant. A system that posts nonurgent interrupts for later handling is a better choice for real-time work.

(6) Are disk operations handled on a DMA basis? The primary disadvantage of floppy disks on many small systems is not that the data transfer rate is slow; it is that the CPU is prevented from doing other useful work while disk transfers occur. Without DMA capability, it is impossible to dump one trial's data while the next trial is proceeding. DMA controllers exist for floppy disk systems and are necessary if any substantial amount of data transfer is to occur during the course of an experiment.

(7) How many concurrent DMA channels may be active? This depends on the design of the processor, the memory, the memory management system, and the software. The answer determines how many essentially continuous I/O processes may be occurring together. For example, to continuously generate a stimulus with a D/A converter, record a tracking response with an A/D converter, and dump the data to disk or tape may require three DMA channels active simultaneously. The goal here is not necessarily high data throughput rate; instead, it is the maintenance of accurate timing on the two analog channels during the CPU activity needed to service the changeover between blocks on the storage device.

WHY MULTIPROGRAMMING?

Manacher (1967) has introduced the distinction between "hard" and "soft" real-time systems. In a hard system, responses to events must be reliably made within specified deadlines; in a soft system, statistical distributions of acceptable response times are specified. An example of the latter constraint might occur in an on-line reservations system, where 80% of the queries would be answered within 10 sec, the remaining 20% within 30 sec. The distinction is closely related to that between peak response capacity and average response capacity of a computer system. Laboratory systems, requiring the accurate control and measurement of real-world events, are inherently hard systems, concerned with timing deadlines that can be met reliably. If the processes of generating stimuli, collecting responses, analyzing the relationships, and communicating with external peripherals are viewed as sequentially alternate tasks, then the achievable paradigms are limited by the worst-case performance of any link in the chain.

Commercial multiprogramming software developed from a desire to match the average capacity of parts of a computer system to demands upon that capacity. If any one program uses only 10% of the available CPU

and disk access time, then it is reasonable to execute 10 such programs concurrently, giving each the required average response from the system resources. With the addition of priority relationships among the tasks, such multiprogramming systems are advertised as sharing a CPU and disks among various users, avoiding duplication of services, and ensuring that high-priority real-time tasks are not inhibited by unrelated lower priority real-time or background tasks. Such systems may also solve the real but secondary problem of allowing editing, compilation, and data analysis to execute on the same computer used to collect the data, at a time when both experimenters and programmers want access. In other words, the usual arguments for multiprogramming are largely economic ones, variants of "why buy two when one will do?"

In the laboratory, multiprogramming systems can perform a much more valuable service: They can match the "hard" demands of foreground tasks to the average capacity, rather than the peak capacity, of background tasks. The effects of worst-case assumptions about CPU, disk, or terminal speeds can be diffused over many iterations of a foreground process, allowing that foreground process to proceed at an accurately timed, reliable rate that is not obtainable with single-task-stream programming techniques.

Although even the slowest of current computers is fast by human standards, none is infinitely fast. Especially when combined with high-overhead user

languages such as BASIC, which make heavy use of floating-point software in all operations, the CPU time consumed by a computer can be appreciable compared to the timing intervals involved in many typical experiments. When the relatively slow speed of these languages makes us consider the need for a faster computer, we should seriously ask which dimension of speed is our problem—overall throughput, peak response capability, or lack of uninterrupted processing time. For most cases of the third problem, and many cases of the second problem, multiprogramming software is a viable alternative to faster hardware or to a less convenient host language.

REFERENCES

- DJIKSTRA, E. W. The structure of the 'THE' multiprogramming system. *Communications of the ACM*, 1968, **11**, 341-346.
- KAPLAN, H. L. Clock-driven FORTRAN task scheduling in a multiprogramming environment. *Behavior Research Methods & Instrumentation*, 1977, **9**, 176-183.
- KAPLAN, H. L. An output spooling system for continuous data-logging paradigms. *Behavior Research Methods & Instrumentation*, 1978, **10**, 285-290.
- LEE, J. A. N. Definition of "semaphore." In A. Ralston & C. L. Meeks (Eds.), *Encyclopedia of computer science*. New York: Petrocelli/Charter, 1976.
- MANACHER, G. K. Production and stabilization of real-time task schedules. *Journal of the ACM*, 1967, **14**, 439-465.
- POLSON, P. G. Symposium: Hardware and software. *Behavior Research Methods & Instrumentation*, 1977, **9**, 162-163.